
Microsoft®
Research
Faculty Summit
Microsoft

Phoenix Tools Infrastructure: On-going Research From 3 Early Adopters

Hoi Vo
Assistant Director
Binary Technologies Team
Microsoft Corporation

Phoenix Early Adopters

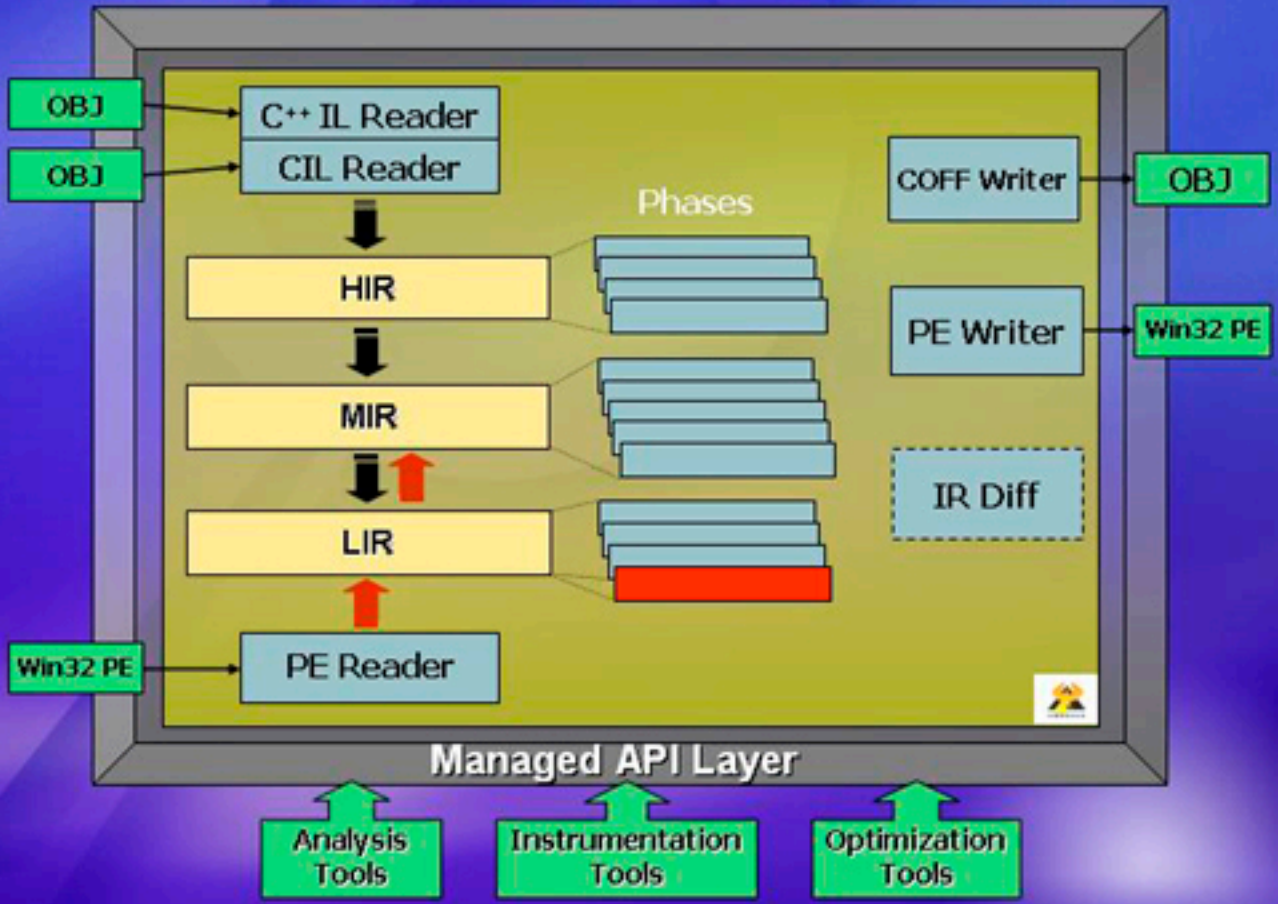
Sponsored by the Binary Technologies (BiT) team

Agenda

- Status on the Phoenix Tools Infrastructure
- Binary Technologies (BiT) projects
- On-going Early Adopters projects
- Q&A

Phoenix Tools Infrastructure

Current Status



Binary Technologies Team
Current Projects

- < DemoFest booth #17>
- Static analysis (focus on CIL)
  - Interprocedural variable tracker
  - Symbolic execution simulator
- Profiling
  - Basic block
  - Edge flow
- Performance analyzer

Phoenix Early Adopters

On-going Projects

Professor Rajiv Gupta
University of Arizona

Precise Dynamic Program Slicing

Professor Michael D. Smith
Harvard University

Graph Coloring Register Allocation

Professor Brad Calder
UCSD

Detecting Program Phases

Using Phoenix for Profiling Research

Rajiv Gupta

Sriraman Tallam & Xiangyu Zhang

The University of Arizona

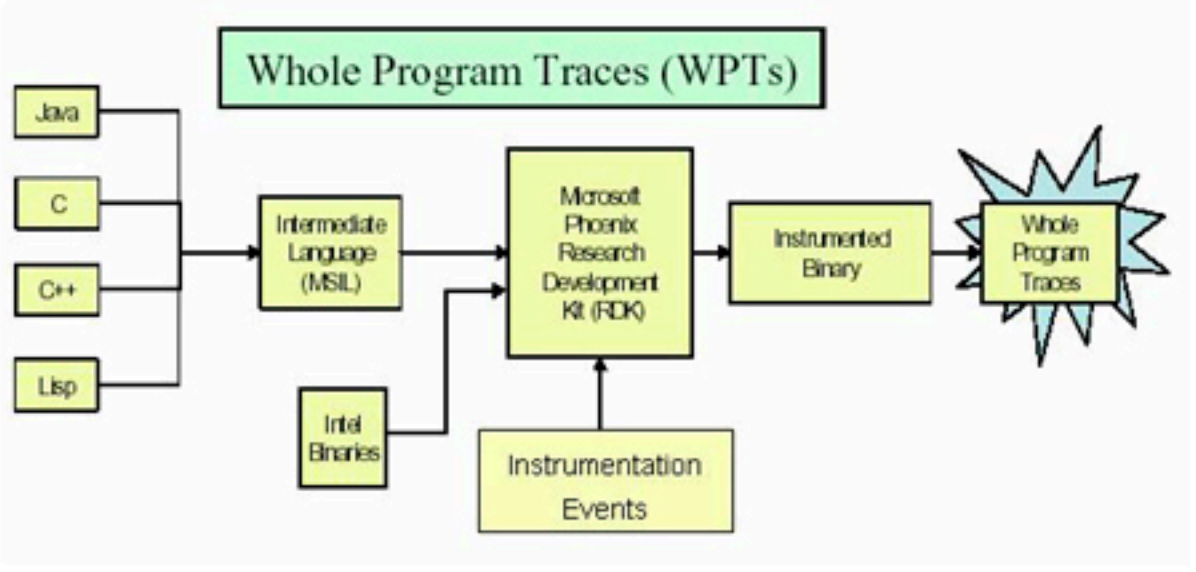
Dependence Profiling

- Precise Dynamic Dependence History
  - Data and Control History
  - Dynamic Dependence Graph
  - Precise Dynamic Program Slicing
    - Debugging, Testing, Criticality, Redundancy ….
- Approximate Dynamic Dependence History
  - Dependence Edge Frequencies
  - Dependence Chain Frequencies
  - Practical Dynamic Program Slicing

Unified Representation: Whole Program Traces

- Static Program Representation
  - **Control flow graph**
  - **Program Dependences**
    - Control dependences
    - Data dependences
- Dynamic Profile Representation
  - **Designed for Analysis**
    - Annotates static program representation
    - Related information can be easily accessed
  - **Comprehensive**
    - Control flow
    - Addresses and values
    - Data and control dependences
  - **Extensible**
    - New types of annotations can be easily added
  - **Compact**
    - Design compaction techniques

Project Goal



Profiling Using Phoenix

- Used Phoenix to do basic block profiling to identify hot basic blocks.
  - < 200 lines of programming.
- Used Phoenix to instrument the binaries to print out load address traces.
  - < 200 lines of programming.
- Most of the code was for reading and parsing the binaries which was the same for both.
  - Real code was < 40 lines.

Transformation and Profiling Using Phoenix

Profiling data dependence chains: program is transformed to establish relationship between data dependence chains and paths frequencies.

- C# benchmark → MSIL binary.
- MSIL binary → Phoenix Lowest level IR.
- Transform IR → Code Duplication.
- Instrument IR to Collect Path Profiles.
- Write out the binary.

Experience with Phoenix

- Getting Started
  - Sample programs and code snippets available.
- Documentation
  - Phoenix Class structure was a good reference.
- Source Code
  - Phoenix code is buggy. Access to source code was very useful. Buggy code can be avoided by using alternatives.
- IR Class
  - Well-designed and intuitive. Moving instructions around & adding/removing instructions are easy to do.
- Alias Analysis
  - Hard to get correspondence between MSIL and LIR.

Phoenix Early Adopters

On-going Projects

Professor Rajiv Gupta
University of Arizona

Precise Dynamic Program Slicing

Professor Michael D. Smith
Harvard University

Graph Coloring Register Allocation

Professor Brad Calder
UCSD

Detecting Program Phases

Project Goals

- Implement a Phoenix phase based on our generalized algorithm for graph-coloring register allocation
- Use Phoenix to evaluate the compile-time efficiency and run-time effectiveness of the algorithm

Chaitin's Assumptions

- **Interchangeable**

  Registers are equally suitable in any program context

- **Independent**

  Writing to one register cannot change the value of another

Example: x86

Register classes

C_{EX} : { eax,ebx,ecx,edx }

C_{X} : { ax,bx,cx,dx }

C_{LH} : { al,ah,…,dl,dh }

C_{EI} : { esi,edi }

C_{I} : { si,di }

C_{EXI} : $C_{EX} \cup C_{EI}$

C_{XI} : $C_{X} \cup C_{I}$

Alias sets

alias(eax) = {eax, ax, al, ah}

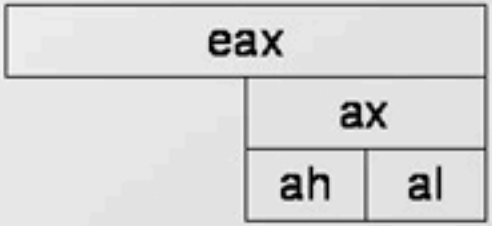
alias(ebx) = {ebx, bx, bl, bh}

…

alias(al) = {eax, ax, al}

alias(ah) = {eax, ax, ah}

…



Our Generalized Solution

- Directly supports
  - simultaneous allocation of overlapping classes
  - register aliasing
- Maintains efficiency and general structure of original graph-coloring formulation
- Makes targeting a machine simple
  - define register classes and alias sets

Big Picture

- Keep interference graph
  - a node represents a register candidate
  - an edge connects candidates that interfere
  - interfering register candidates cannot be assigned to registers whose alias sets overlap
- Change colorability criterion
  - Chaitin's: $degree_n < k$

Trivial Colorability

$degree_n < k$

maximum number of these available registers that could be consumed by a coloring of *n*'s neighbors

number of registers available to *n*

$squeeze_n < |N|$

$Squeeze_n$

- Given a class tree with root vertex R:

$$squeeze_n = Z_n(R)$$

where

$$Z_n(v) = \min\left(bound(N, v), rawZ_n(v)\right)$$

$$rawZ_n(v) = \sum_{C \in classes(v)} worst^{degree_n(C)}(N, C) + \sum_{v' \in children(v)} Z_n(v')$$

Class Tree for x86

Register classes

C_{EX} : { eax, ebx, ecx, edx }

C_{X} : { ax, bx, cx, dx }

C_{LH} : { al, ah, …, dl, dh }

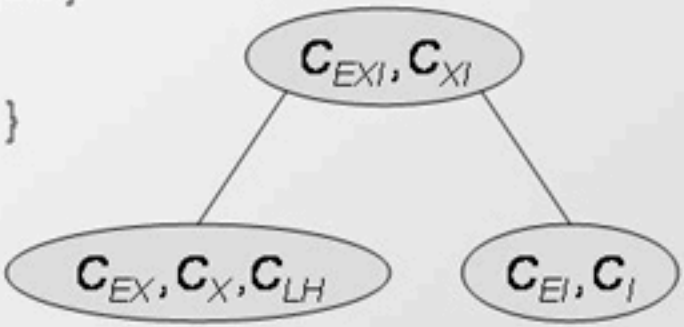
C_{EI} : { esi, edi }

C_{I} : { si, di }

C_{EXI} : $C_{EX} \cup C_{EI}$

C_{XI} : $C_{X} \cup C_{I}$

Class Tree



Class trees are typically small

Experience

1. Phoenix, in general
2. Encapsulating target specifics
3. Replacing an important phase
4. Using the Phoenix API
5. Working with managed code
6. Observing phase performance

Phoenix LIR ≈ Machine SUIF

- Common goals
  - precise manipulation of machine instructions
  - ease of retargeting
- Remarkably similar IR
  - abstraction of machine instructions
  - explicit operands carry data dependence
  - integrated CFG describes control flow
  - extensible annotations cover special cases

Encapsulating Target Specifics

- API gives access to target attributes
  - register file characteristics
  - instruction properties
- Implementation derived from machine description
- General-purpose lowering API
  - target-independent creation of target-specific code

Replacing a Phase

- Harder than you might think
  - target independence
  - understanding the contract of the phase
- Phoenix provides "plumbing"
- Additional guidance could come from
  - more contract documentation
  - a cookbook with templates and example phases

Using the Phoenix API

- Did not extend the IR, but …
- Defined new data structures
  - interference graph and class tree, extending graph and collection classes
- Used an existing analysis: liveness

Working with managed code

- Phoenix plug-in model requires a managed phase
- Converting our implementation from C++ to managed C++ was more involved than expected
- A managed environment seems like a good match for our work

Performance

This is where we are today. We have a working allocator. We're setting up and running experiments.

Phoenix Early Adopters

On-going Projects

Professor Rajiv Gupta
University of Arizona

Precise Dynamic Program Slicing

Professor Michael D. Smith
Harvard University

Graph Coloring Register Allocation

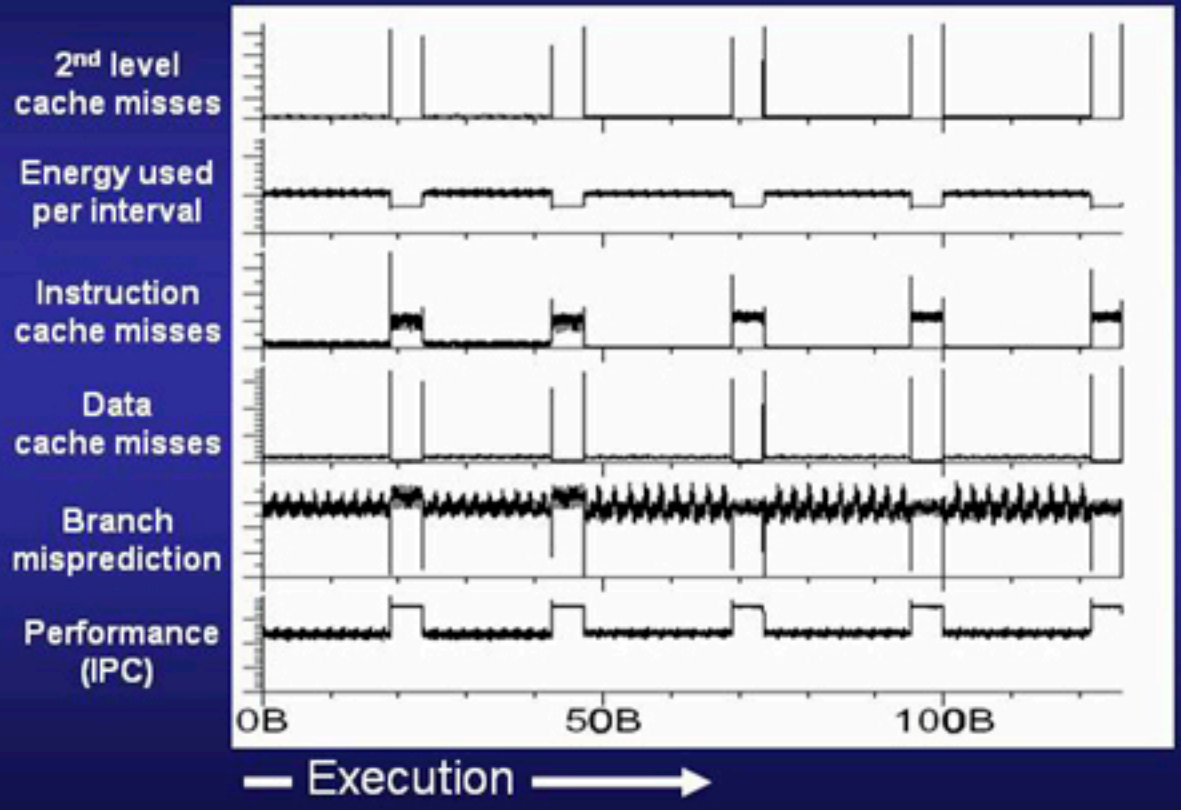
Professor Brad Calder
UCSD

Detecting Program Phases

Phase Research Motivation

- **Hypothesis:** *Programs have repeatable time-varying behavior*
- **Goal:** Automatically identify what parts of a program's execution have similar behavior
  - Guide simulation
  - Guide adaptive hardware or software optimizations
  - Guide OS scheduling
- **Challenge:** To make sense of a sea of data and extract important information

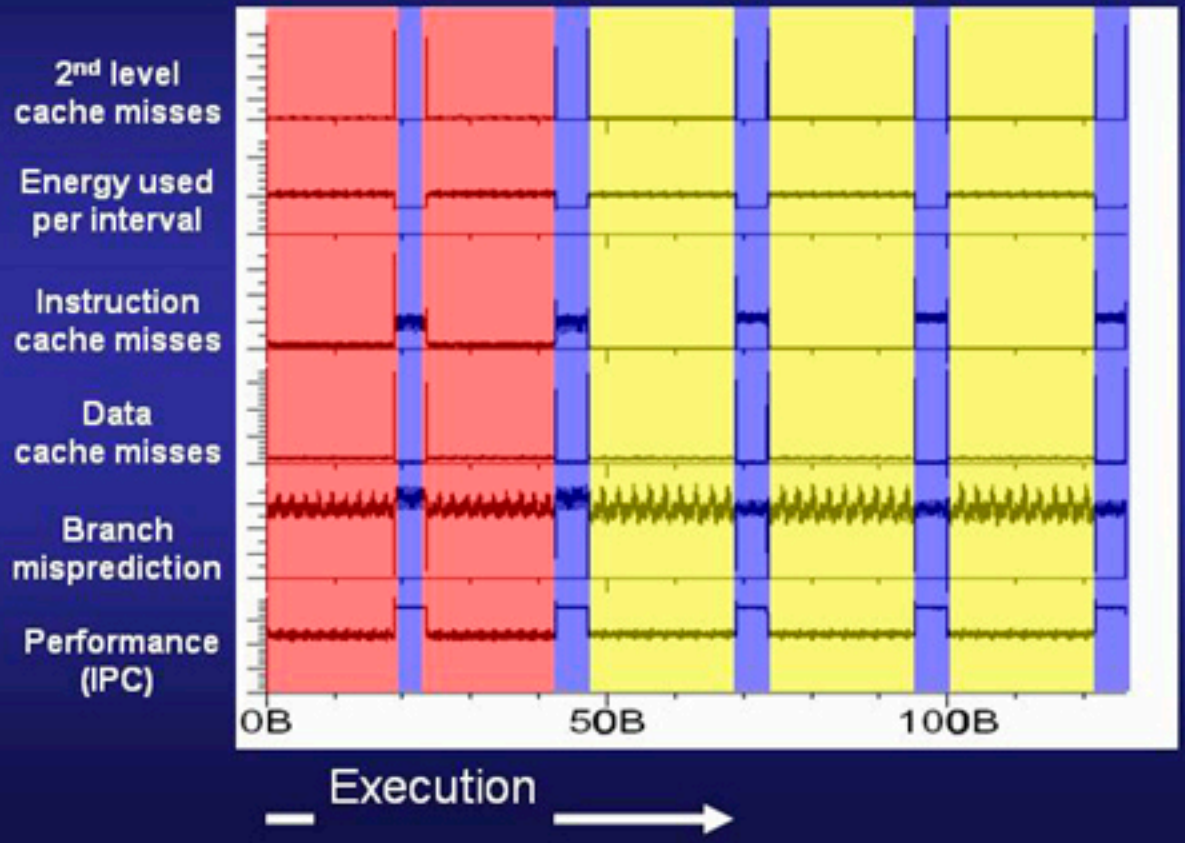
Time Varying Behavior of GZIP



Definitions

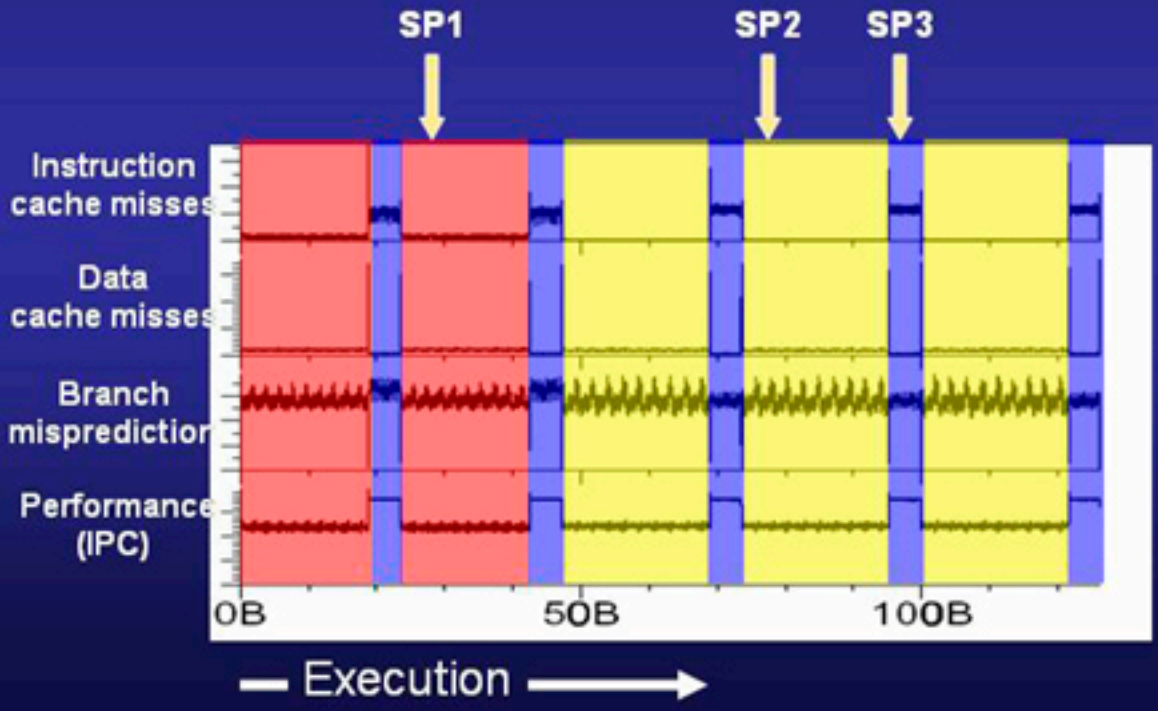
- An interval is a selection of contiguous instructions in program execution order
  - Think of it as a slice in time
- A phase in a program is a set of intervals that are similar to each other
  - The program should have similar behavior (no matter how it is measured) in all intervals of a single phase

Behavior of GZIP



Representative Points - GZIP

- Pick a representative sample from each phase to guide analysis and optimization



You are what you execute

- **Goal** - track behavior of a program
  - Behavior **caused by** the path through code
- **How** - Track the code that is executing
  - Detect changes and similarities in code
- **Code Signatures**
  - Create a vector equal in dimension to number of basic blocks in program
  - Profile for each interval # times each BB executed
  - Subtract vectors to determine similarity
    - 0 means identical, 2 means no overlap in execution

Goals for Using Phoenix

- Create Program-Level Phase Analysis
  - ISA, Compiler and OS independent
    - x86, IA64, ARM, etc…
  - Map the phase analysis back to the source code
- Examine Phase Behavior in Multi-Threaded Progams

Traversing the Binary


```
// Find pointer to analysis routine to call
PhxString dll = L"phxBBtrack.dll";
Phx::Syms::ImportSym * trackSym =
     AddImport(module, dll, L"_TrackBB@8");
Phx::Types::Type * trackType =
     GetExternalFuncType(module, funcCalls::TrackBB);

// Traverse all of the functions
foreach(Phx::IrUnit *, irUnit, module->IrUnitList) {
     // Build the IR for the function
     Phx::PE::ReaderPhase::RaiseIR(irUnit, false);
     Phx::IR::Instr * realInstr = func->FirstInstr;
     // Traverse all of the instructions
     foreach(Phx::IR::Instr *, instr, realInstr->Next) {
        ...
       // Add calls at the start of the basic block
       CreateCall(func, instr, blockID, InstrCount,
                  trackSym, trackType);

     } }
```

Inserting Instrumentation Call

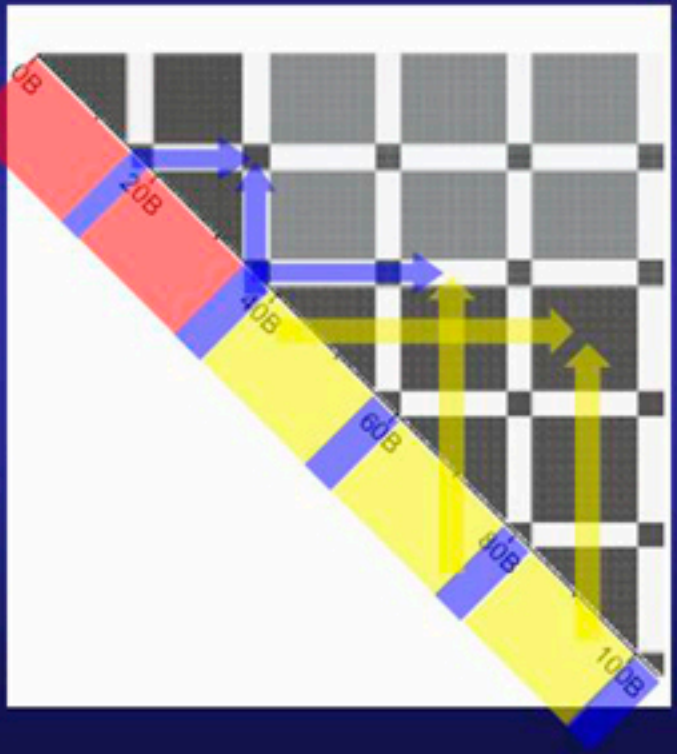

```
// Create the MemOpnd for an indirect call through the IAT
Phx::IR::MemOpnd * opndCallTgt =
   Phx::IR::MemOpnd::New(funcLifetime, type->PrimaryField,
                         sym->IATSym, NULL, 0, align, tag);

// Create a new call instruction
Phx::IR::CallInstr * call = Phx::IR::CallInstr::New(
             funcLifetime, Phx::Opcode::Call, opndCallTgt);

// Append the Operands for the parameters...
Phx::Types::Type * uintType = func->TypeTable->UInt32Type;
Phx::IR::ImmOpnd * op1 = Phx::IR::ImmOpnd::New(funcLifetime,
                                    uintType, op1val);
call->AppendSrc(op1);
Phx::IR::ImmOpnd * op2 = Phx::IR::ImmOpnd::New(funcLifetime,
                                      uintType, op2val);
call->AppendSrc(op2);

// Add the call before the instruction
instr->InsertBefore(call);
```

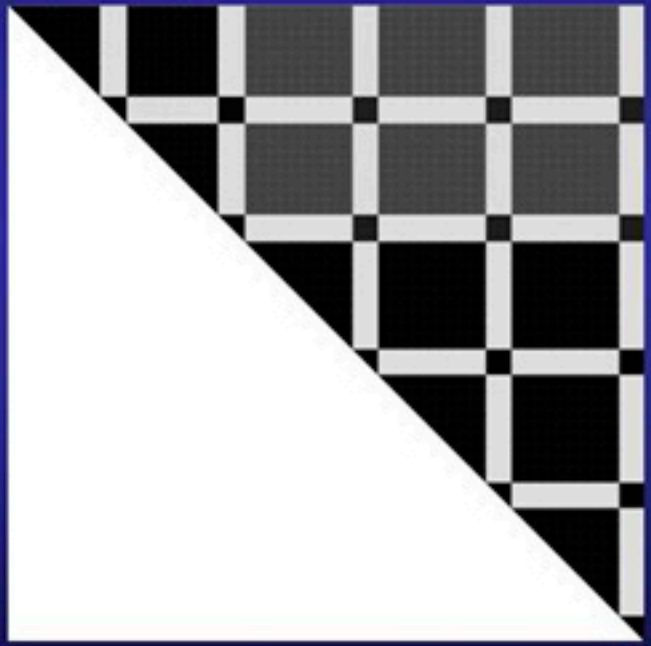
Basic Block Similarity Matrix



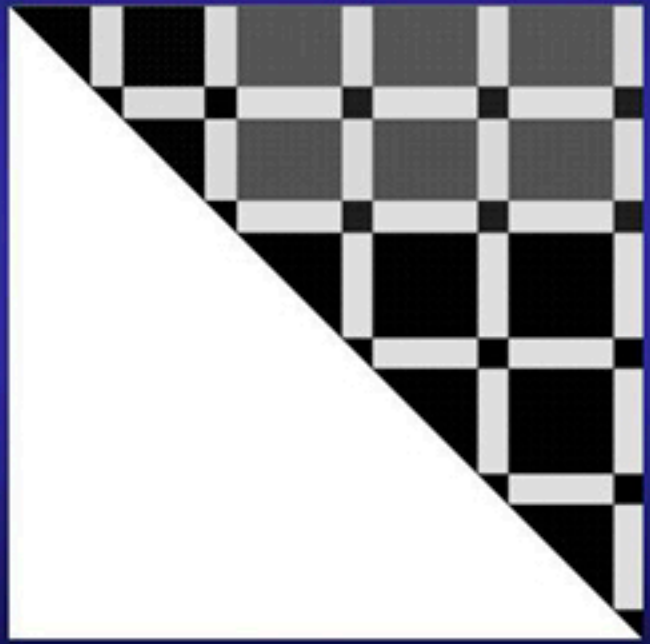
- Compare N^2 intervals
- Executed Instructions on Diagonal axis
- To compare 2 points go horizontal from one and vertically from the other
- Darker points indicate similar vectors
- Clearly shows the phase-behavior without any detailed simulation

GZIP BB Similarity Graph

- Comparing the similarity of basic block vectors between two compilers and ISAs



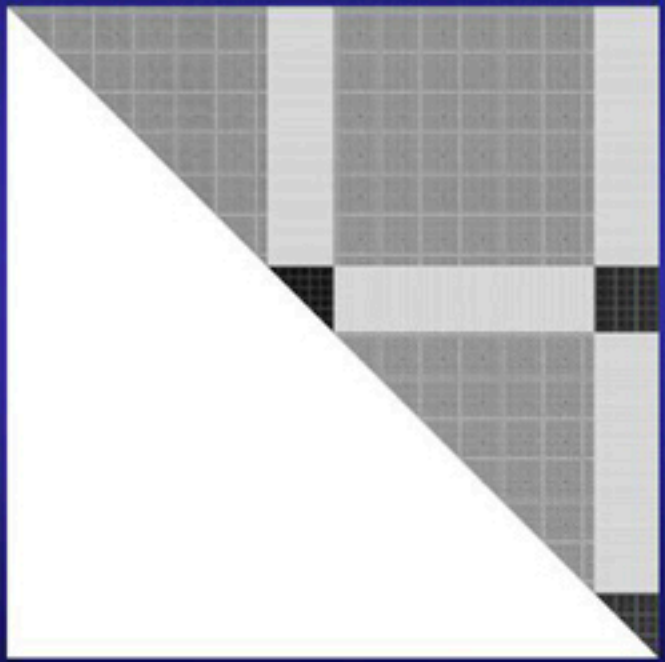
Alpha - ATOM



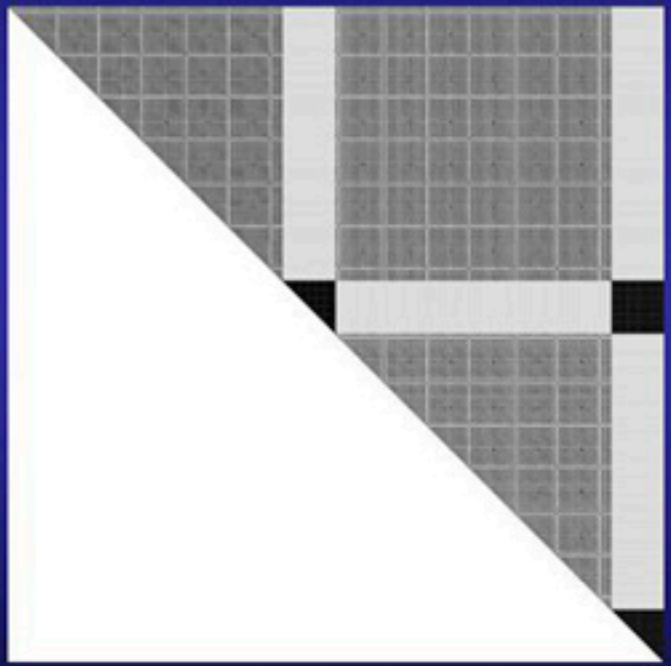
x86 - Phoenix

BZIP BB Similarity Graph

- Comparing the similarity of basic block vectors between two compilers and ISAs



Alpha - ATOM



x86 - Phoenix

Phoenix Phase Summary

- Found:
  - Possible to identify same phase behavior across different ISAs and compilers
- Next:
  - Automatically map the phase information back to the source code, and provide Source Level Simulation Points

Phoenix Early Adopters

On-going Projects

Professor Rajiv Gupta
University of Arizona

Precise Dynamic Program Slicing

Professor Michael D. Smith
Harvard University

Graph Coloring Register Allocation

Professor Brad Calder
UCSD

Detecting Program Phases


Q&A

Microsoft®

Your potential. Our passion.™